Report di Penetration Testing: Buffer Overflow (System Exploitation)

Data: 26 Gennaio 2026

Esercizio: Day 3 - System Exploit BOF

Target: Codice C BW_D3_BOF.c

1. Introduzione e Obiettivi

L'attività richiesta consiste nell'analizzare un programma scritto in linguaggio C, comprenderne il funzionamento e modificarlo per dimostrare una vulnerabilità di tipo Buffer Overflow (Stack-based). L'obiettivo finale è causare un errore di segmentazione (crash) sovrascrivendo aree critiche della memoria.

2. Analisi Statica: Come funziona il programma

Prima di eseguirlo, leggiamo il codice C (BW_D3_BOF.c) per capire cosa fa.

Struttura del codice:

1. **Dichiarazione:** Viene dichiarato un array (vettore) di interi chiamato vector di dimensione fissa **10** (int vector[10]).
2. **Input:** Il programma chiede all'utente di inserire **10** numeri interi tramite un ciclo for che va da 0 a 9.
3. **Echo:** Ristampa a video i numeri appena inseriti (per conferma).

4. **Ordinamento:** Utilizza un algoritmo chiamato **Bubble Sort** (i due cicli for annidati con la variabile swap_var) per mettere in ordine crescente i numeri.
5. **Output Finale:** Stampa il vettore ordinato.

Ipotesi di funzionamento: Il programma è scritto per gestire esattamente 10 numeri. Se l'utente inserisce 10 numeri, il programma funzionerà perfettamente, li ordinerà e terminerà con successo (return 0).

3. Fase 1: Riproduzione in Laboratorio

Il programma è stato compilato ed eseguito nella sua forma originale per verificarne il corretto funzionamento.

- **Compilazione:** gcc BW_D3_BOF.c -o programma_sicuro
- **Test:** Inserendo 10 interi, il programma li ordina e termina con successo (Exit code 0). L'ipotesi iniziale sul funzionamento era corretta.

4. Fase 2: Creazione dell'Exploit (Modifica del Codice)

Per simulare la vulnerabilità (mancanza di controllo sull'input), è stato modificato il sorgente alterando il limite del ciclo for.

Modifica Apportata: Il ciclo di input è stato esteso da i < 10 a i < 500.

Questo permette all'utente di inserire 500 numeri in un contenitore che ne può ospitare solo 10.

Compilazione Vulnerabile: Per facilitare l'exploit in un ambiente moderno (Kali Linux), sono state disabilitate le protezioni dello stack (canary) durante la compilazione:

gcc -fno-stack-protector -z execstack -no-pie BW_D3_BOF.c -o bof_vulnerabile

5. Fase 3: Exploitation e Analisi Memoria

Tentativo 1: Input Ripetitivo (Fallito)

Inizialmente è stato tentato un attacco inserendo sempre lo stesso numero (1) per 500 volte usando il comando:

python3 -c "print('1\n' * 200)" | ./bof_vulnerabile

- **Risultato:** Il programma **non è andato in crash**.
- **Analisi Tecnica:** Analizzando il comportamento, si è scoperto che la variabile contatore i (che gestisce il ciclo) si trova nello stack subito dopo il vettore vector. Quando l'input ha superato la decima posizione, ha sovrascritto i con il valore 1. Poiché 1 < 500, il ciclo è "tornato indietro", ricominciando a scrivere dall'inizio del vettore all'infinito, senza mai avanzare verso l'indirizzo di ritorno.

Tentativo 2: Input Sequenziale (Successo)

Per evitare di resettare il ciclo, è stato necessario inviare una sequenza di numeri crescenti (0, 1, 2, 3...). In questo modo, quando l'overflow sovrascrive la variabile i, le assegna un valore maggiore, permettendo al ciclo di avanzare e corrompere le aree successive della memoria.

python3 -c "for x in range(500): print(x)" | ./bof_vulnerabile

6.BONUS

Cosa cambieremo:

1. Aggiungiamo una variabile scelta per il menù.
2. Aggiungiamo una variabile limite_loop che cambierà dinamicamente:
    - 10 se l'utente vuole il programma sicuro.
    - 500 se l'utente vuole il programma che crasha.
3. Inseriamo un controllo if per gestire la scelta.

Quando compiliamo il programma usiamo nuovamente il codice:

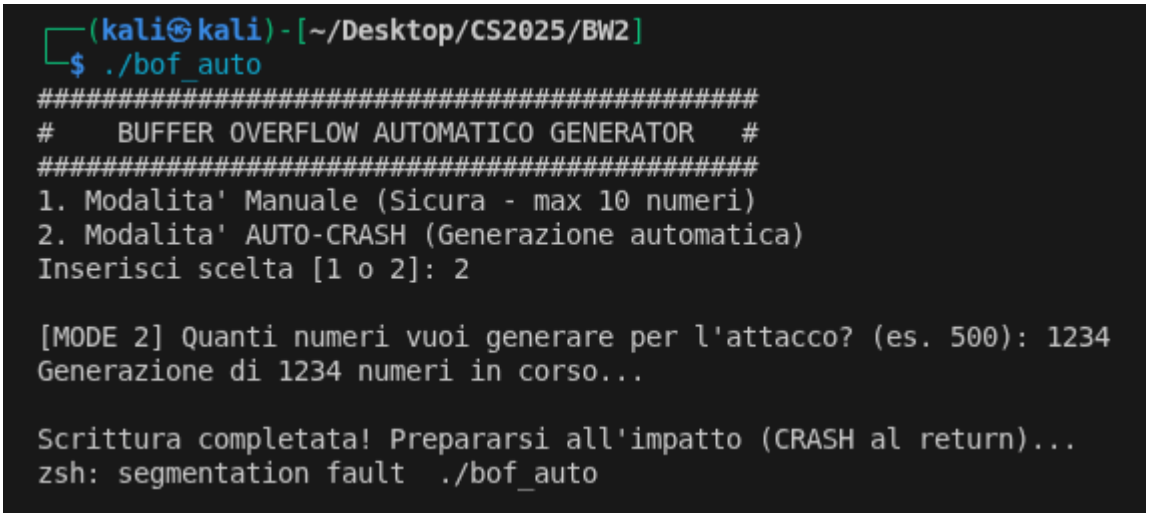
gcc -fno-stack-protector -z execstack -no-pie BW_D3_BOF.c -o bof_auto

Per disabilitare le sicurezze

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main () {

    int vector[10], j, k;
    int swap_var;
    int scelta;

    // Variabili statiche per sopravvivere all'overflow
    static int i;
    static int limite_loop;

    printf("########################################\n");
    printf("#    BUFFER OVERFLOW AUTOMATICO GENERATOR   #\n");
    printf("########################################\n");
    printf("1. Modalita' Manuale (Sicura - max 10 numeri)\n");
    printf("2. Modalita' AUTO-CRASH (Generazione automatica)\n");
    printf("Inserisci scelta [1 o 2]: ");

    if (scanf("%d", &scelta) != 1) return 1;

    if (scelta == 1) {
        printf("\n[MODE 1] Inserisci 10 numeri manualmente:\n");
        limite_loop = 10;

        for (i = 0; i < limite_loop; i++) {
            printf("[%d]: ", i+1);
            scanf("%d", &vector[i]);
        }
    }
    else if (scelta == 2) {
        printf("\n[MODE 2] Quanti numeri vuoi generare per l'attacco? (es. 500): ");
        scanf("%d", &limite_loop);

        printf("Generazione di %d numeri in corso...\n", limite_loop);

        // Ciclo di distruzione memoria
        for (i = 0; i < limite_loop; i++) {
            vector[i] = i;
            // Commento la print riga per riga per rendere l'output più pulito
            // printf("Scrittura vector[%d]... \r", i);
        }
        printf("\nScrittura completata! Prepararsi all'impatto (CRASH al return)...\n");
    }
    else {
        printf("Scelta non valida.\n");
        return 0;
    }

    // --- MODIFICA FONDAMENTALE QUI SOTTO ---
    // Eseguiamo l'ordinamento e la stampa SOLO se siamo in modalità sicura.
    // Se siamo in modalità 2, saltiamo questo blocco e andiamo dritti al crash.

    if (scelta == 1) {

        printf("\n--- Esecuzione Bubble Sort ---\n");
        printf ("Il vettore inserito e':\n");
        for ( i = 0 ; i < 10 ; i++) {
            printf("[%d]: %d\n", i+1, vector[i]);
        }
```

```c
    if (scelta == 1) {

        printf("\n--- Esecuzione Bubble Sort ---\n");
        printf ("Il vettore inserito e':\n");
        for ( i = 0 ; i < 10 ; i++) {
            printf("[%d]: %d\n", i+1, vector[i]);
        }

        // Algoritmo Bubble Sort
        for (j = 0 ; j < 10 - 1; j++) {
            for (k = 0 ; k < 10 - j - 1; k++) {
                if (vector[k] > vector[k+1]) {
                    swap_var = vector[k];
                    vector[k] = vector[k+1];
                    vector[k+1] = swap_var;
                }
            }
        }

        printf("\nIl vettore ordinato e':\n");
        for (j = 0; j < 10; j++) {
            printf("[%d]: %d\n", j+1, vector[j]);
        }
    }

    // --- IL MOMENTO DELLA VERITÀ ---
    // In modalità 2, il programma arriva qui, prova a eseguire 'return'
    // usando l'indirizzo di memoria che hai sovrascritto e CRASHA.
    return 0;
}
```

Eseguendo il programma in modalità sicura non ci sono problemi, per non dover scrivere manualmente 500 numero in modalità vulnerabile il programma chiede quanti numeri vogliamo, poi va ad eseguirsi andando in buffer overflow



7. Conclusioni

L'attività ha dimostrato con successo un **Buffer Overflow**. Sfruttando la mancanza di controlli sulla lunghezza dell'input (Bounds Checking), è stato possibile:

1. Uscire dai limiti dell'array vector.
2. Manipolare il flusso di esecuzione del programma (sovrascrivendo la variabile i).
3. Corrompere l'indirizzo di ritorno causando il crash del processo.

Raccomandazioni di Sicurezza: Per prevenire queste vulnerabilità in scenari reali, è necessario:

- Utilizzare sempre funzioni sicure che limitano l'input (es. fgets invece di gets, o specificare la lunghezza in scanf).

- Verificare sempre che l'indice di scrittura non superi la dimensione allocata (sizeof).
- Mantenere attive le protezioni del compilatore (Stack Canaries, ASLR).